Ch7. Linear Lists – Simulated Pointers

SNU
IDB Lab.

Bird’s-Eye View

- Ch.5 ~ Ch.7: Linear List
  - Ch. 5 – array representation
  - Ch. 6 – linked representation
  - Ch. 7 – simulated pointer representation
    - Simulated Pointers
    - Memory Management
    - Comparison with Garbage Collection
    - Simulated Chains
    - Memory Managed Chains
    - Application: Union-Find Problem

※ In succeeding chapters - matrices, stacks, queues, dictionaries, priority queues

- Java’s linear list classes
  - java.util.ArrayList
  - Java.util.Vector
  - java.util.LinkedList

Bird's-Eye View

- Simulated-pointer representation
  - What if we want to have linked structures on disk
  - What if we want to have user-defined pointers instead of Java references
  - Simulated pointers are represented by integers rather than by Java references
- To use simulated pointers
  - Must implement our own memory management scheme: a scheme to keep track of the free nodes in our memory (i.e., array of nodes)

Table of Contents

- Simulated Pointers
- Memory Management
- Comparison with Garbage Collection
- Simulated Chains
- Memory Managed Chains
- Application: Union-Find Problem

The Need for Simulated Pointers

- Memory allocation via the Java new method
  - Automatically reclaimed by garbage collection

- Java references are internal memory addresses and not addresses of disk memory

- Java references (pointers) cannot be used for
  - Disk storage management
  - Data structure backup
  - External data structures

- Solution
  - Simulated pointers
  - User defined memory allocation and deallocation

Disk Storage Management

- Operating System’s disk management
- Disk is partitioned into blocks of a predetermined size
  - So called “block size” (say 32KB)
- These blocks are linked together from a chain
- Each chain entry is made in the file allocation table (FAT)
- The address of each disk block is different from main memory address
- Simulated pointers make easy the process

Data Structure Backup

- What if you want to work on a chain of student grades next week?
  - Serialization: the process of writing every element of the data structure in some sequential order into a file
  - Deserialization: read back the serialized version from the file and reconstruct the chain in memory

- During deserialization we need to capture the pointer information to reconstruct the linked structure

- Simulated pointers make easy the process
- So called, Persistent Data Structure

External Data Structures

- Data structures with pointers for the data on a disk
- B+ tree index (will soon be covered)
  - Leaf nodes of B+ tree are pointing the records on a disk

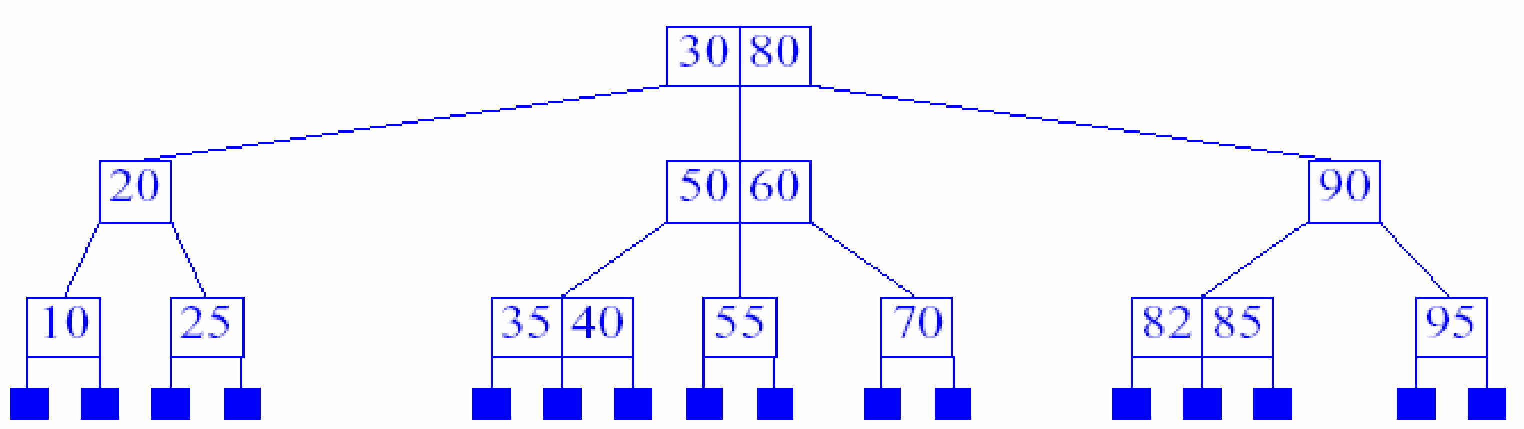
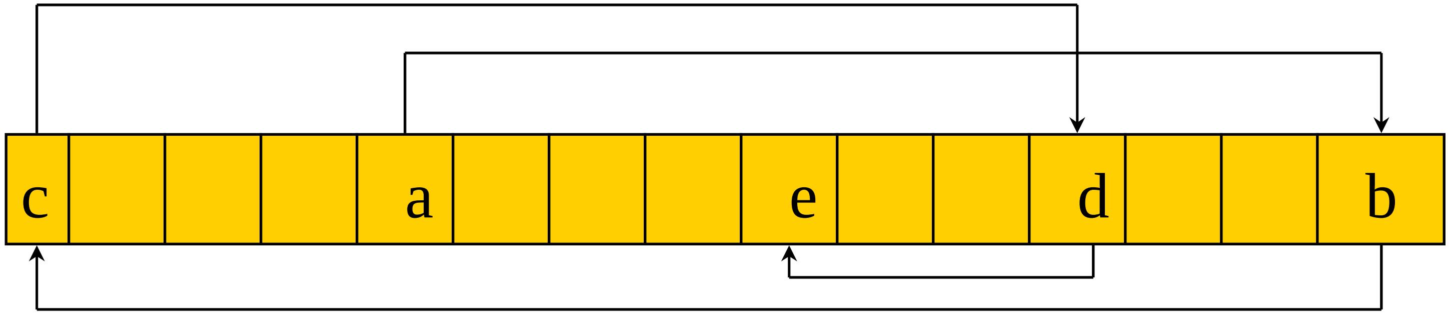


Figure 16.25 A 2-3 tree or B-tree of order 3

Simulating Pointers

- How to simulate pointers in internal memory?
  - By implementing linked lists using an array of nodes
  - By simulating Java references by integers that are indexes into this array
- Useful for backup and recovery of data structure
  - To backup, we need merely back up the contents of each node as it appears from left to right in the node array
  - To recover, we read back the node contents in left-to-right in the node array



- Each array position has an element field and a next field (type int)

Node Representation

```
class SimulatedNode
{ // package visible data members
   Object element;
   int next;
  // package visible constructors
   SimulatedNode() { };
   SimulatedNode(int next)
                  {this.next = next;}
}
```

- ChainNode’s next: java reference
- SimulatedNode’s next: int type

How It All Looks?

- Initially, the nodes in the available space were all empty nodes
- Allocate nodes & store “a b c d e”
- Free nodes are members of a linked list
- In-use nodes are also members of a linked list

next	11				14				-1			8			0
element	c				a				e			d			b
	0	1	2	3	4	5			8			11			14

firstNode = 4

Table of Contents

- Simulated Pointers
- Memory Management
- Comparison with Garbage Collection
- Simulated Chains
- Memory Managed Chains
- Application: Union-Find Problem

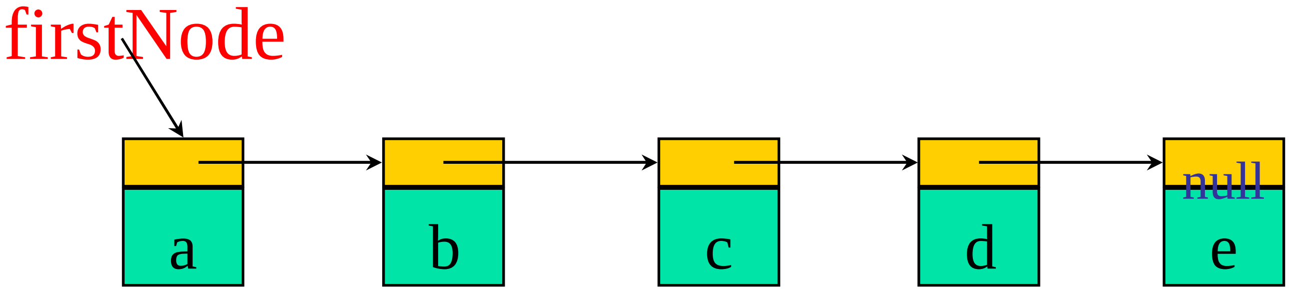
Memory Management using SP

- Memory management
  - Create a collection of nodes (a storage pool): node[0:numOfNodes-1]
  - Allocate a node as needed: allocateNode()
  - Reclaim nodes that are no longer in use: deallocateNode()

- In our simple memory management scheme, nodes that are not in use are kept in a storage pool

- Memory management with different sizes is rather complex

Storage Pool (same size nodes)



- Maintain a chain of free nodes
- In the beginning, all nodes are free
- Allocate a free node from the front of chain
- Add node that is freed (deallocated) to the front of chain

The Class SimulatedSpace1

```
/** memory management for simulated pointer classes */
package dataStructures;
import utilities.*;
public class SimulatedSpace1
{ // data members
   private int              firstNode;
   SimulatedNode []         node;

  // package visible constructor and other methods here
}
```

Constructor of SimulatedSpace1

** creating the available space list

```
public SimulatedSpace1(int numberOfNodes)
  {  node = new SimulatedNode [numberOfNodes];  // array declaration
     // create nodes and link into a chain: array initializatin
     for (int i = 0; i < numberOfNodes - 1; i++)
                   node[i] = new SimulatedNode(i + 1);
     // last node of array and chain
     node[numberOfNodes - 1] = new SimulatedNode(-1);
     // firstNode has the default initial value 0
  }
```

Array in Java

- Primitive type: declaration & allocation at once

```
node = new int [10]
```

VS

- Complex type: declaration & allocation separately

```
node = new SimulatedNode[10]
for (int i = 0; i < 9; i++)
        node[i] = new SimulatedNode(i+1);
```

Allocate a Node using SP: O(n)

```
public int allocateNode(Object element, int next)
  {// Allocate a free node and set its fields.
     if (firstNode == -1)
     {   // if no more free nodes in the available space list,
         // create and line new nodes (doubling)}

     int i = firstNode;              // allocate first node
     firstNode = node[i].next;   // firstNode points to next free node

     node[i].element = element; // set its fields
     node[i].next = next;
     return i;                          // return the sp of new node
  }
```

Free a Node using SP: O(1)

```
public void deallocateNode(int i)
 {// Free node i.
   // make i first node on free space list
   node[i].next = firstNode;
   firstNode = i;

  // remove element reference so that the space
  // (the referenced “ABC”) can be garbage collected:
   node[i].element = null;
 }
```

The class SimulatedSpace2

- A list for free nodes not been used yet (first1) & used at least once (first2)
- Lazy initialization

```
public int allocateNode(Object element, int next)
  {// Allocate a free node and set its fields.
    if (first2 == -1)  {   // 2nd list is empty
       if (first1 == node.length) {
         // code for doubling number of nodes
        }
       node[first1] = new SimulatedNode(); // lazy initialization
       node[first1].element = element;
       node[first1].next = next;     return first1++; }

    int i = first2; // allocate first node of 2nd chain
    first2 = node[i].next;    node[i].element = element;
    node[i].next = next;      return i;
}
```

Facts of Simulated Pointers

- Can free a chain of nodes in O(1) time (bulk deallocation) when first node f and last node e of chain are known
  - Node[e].next = firstnode;
    firstNode = f;

- If you deal with only in-memory stuff, don't use simulated pointers unless you see a clear advantage to using simulated pointers over Java references

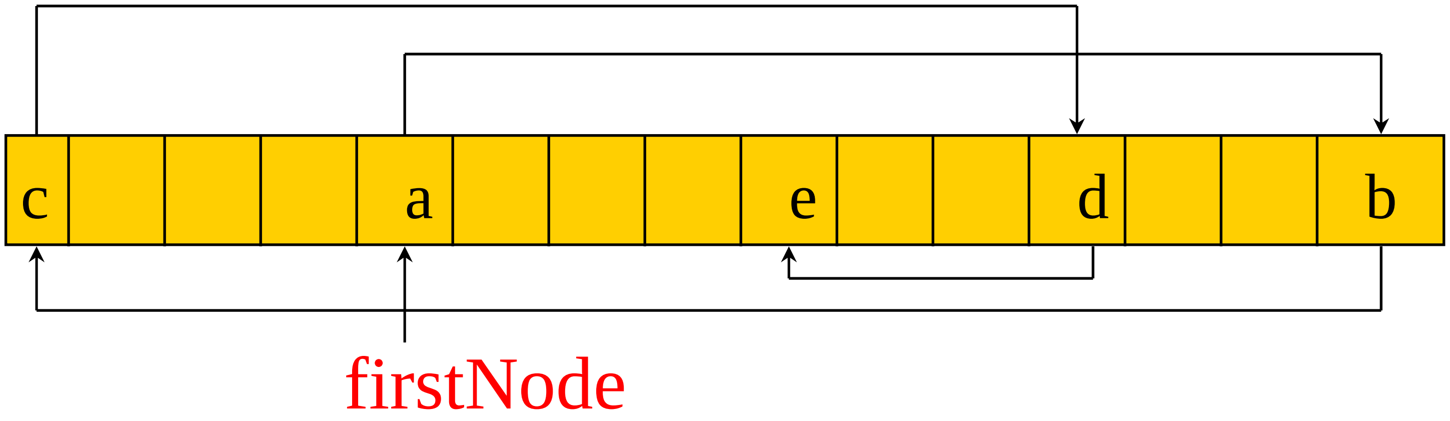
Table of Contents

- Simulated Pointers
- Memory Management
- Comparison with Garbage Collection
- Simulated Chains
- Memory Managed Chains
- Application – Union-Find Problem

Garbage Collection (GC)

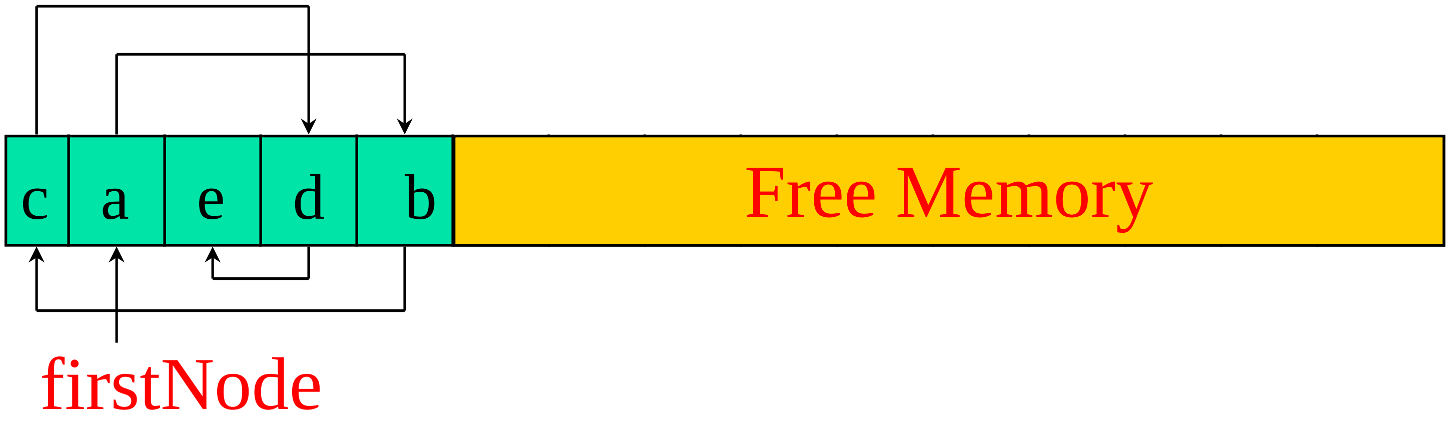
- User’s DeallocateNode vs. System’s Garbage Collection
- GC: The system determines which nodes/memory are not in use and returns these nodes (this memory) to the pool of free storage
- Periodic & Automatic Invokation
- This is done in two or three steps
  - Mark nodes that are not in use
  - Compact free spaces (optional)
  - Move free nodes to storage pool

GC Step 1: Marking



- There is a mark-bit for each node
- Unmark all nodes (set all mark-bits "false")
- Marking: Start at firstNode and mark all nodes reachable from firstNode by setting the mark-bit "true"
- Repeat marking for all reference variables

GC Step 2: Compaction (optional)



- Move all marked nodes (i.e., nodes in use) to one end of memory, and update all pointers as necessary

GC Step 3: Restoring Free Memory

- The storage pool is also a linked list
- Free nodes are linked with the storage pool

- If the reusable nodes can be found by scanning memory for unmarked nodes ➔ return those nodes to the storage pool
- Otherwise (cannot find reusable nodes) ➔ need to put a new single free block into the storage pool

Facts of GC

- Due to automatic GC, programmers doesn't have to worry about freeing nodes as they become free
- However, for garbage collection to be effective, we must set reference variables to null when the object being referenced is no longer needed (still the programmer's responsibility!)
- In general, the actual exec time of deallocateNode is faster than that of GC
  - Garbage collection time is linear in memory size (not in amount of free memory). GC could be expensive!
- Application may run faster when run on computers that have more memory because GC does not need to be invoked frequently
- Sometimes GC wins, sometimes deallocateNode wins depending upon the characteristics of application and the size of given memory

Alternatives to Garbage Collection

▪ malloc()/free()	at C language
▪ new()/delete()	at C++ language
▪ new()/GC	at Java

- By manual “delete()” and “free()”, now free nodes are always in storage pool
- Time to free node by “delete()” and “free()” is proportional to number of nodes being freed and not to total memory size

Table of Contents

- Simulated Pointers
- Memory Management
- Comparison with Garbage Collection
- Simulated Chains
- Memory Managed Chains
- Application: Union-Find Problem

Simulated Chains (Linear List with SP)

- So far, we concerned only the free space management with simulated pointers

- Now, we move to LinearList that use the simulated space S for storing and retrieving the data elements
  - S is declared as a static data member
  - So, all simulated chains share the same simulated space

- Linear List implementations
  - FastArrayLinearList (Array)
    - Get(O(1)), Remove(O(n-k)), Add(O(n-k))
  - Chain (Linked list)
    - Java based & Simulated pointers
    - Get(O(k)), Remove(O(k)), Add(O(k))

- Figure 7.5 (242pp) shows the performances

The Class SimulatedChain

```
public class SimulatedChain implements LinearList
{ // data members
   private int firstNode;
   protected int size;
   public static SimulatedSpace1 S = new SimulatedSpace(10);
  // all simulated chains share S

  //constructors
   public SimulatedChain (int initialCapacity) {
        firstNode = -1;
       // size has the default initial value 0
   }
}
```

The method indexOf()

```
public int indexOf(Object elm) {
   // search the chain for elm;
   int currentNode = firstNode;
   int index = ;  // index of currentNode;
   while (currentNode != -1
     && !S.node[currentNode].element.equals(elem)) {
        currentNode = S.node[currentNode].next; // move to next node
        index++; }
   // make sure we found matching element
   if (currentNode == -1) return -1;
   else return index;
    .........
}
```

Table of Contents

- Simulated Pointers
- Memory Management
- Comparison with Garbage Collection
- Simulated Chains
- Memory Managed Chains
- Application: Union-Find Problem

Memory Managed Chains (1)

- Want to improve the performance of the class Chain (chap 6) without actually using simulated pointers
- Dynamic memory allocation methods such as new usually take a lot more time than memory allocation methods such as allocateNode
- Suppose 10^6 add and 10^6 remove operations are done in a mixed manner and always less than 50 list elements are in the list
  - "new" is invoked 10^6 times in the original Chain class
- If we use allocateNode/deallocateNode
  - Only 50 calls to new() will do with 10^6 times of allocateNode() and deallocateNode() each

Memory Managed Chain (2)

- Even though we do not implement the simulatedChain class, the idea of buffering free nodes is useful!

- Modify the class Chain
  - Add a static data member of type ChainNode :
    - first free node
  - Add a static method deallocateNode :
    - insert a node at the front of the free node chain
  - Add a static method allocateNode :
    - allocates a node from the free node chain (or may call new)
  - Modify Chain.remove :
    - use deallocateNode
  - Modify Chain.add :
    - invoke allocateNode rather than new

Table of Contents

- Simulated Pointers
- Memory Management
- Comparison with Garbage Collection
- Simulated Chains
- Memory Managed Chains
- Application: Union-Find Problem

Equivalence Classes

- The relation R is an equivalence relation Iff the following conditions are true:
  - (a, a) $\in$ R for all a $\in$ U (reflexive)
  - (a, b) $\in$ R iff (b, a) $\in$ R (symmetric)
  - (a, b) $\in$ R and (b, c) $\in$ R ➔ (a, c) $\in$ R (transitive)

- Two elements are equivalent if (a, b) $\in$ R
- Equivalence class
  - A maximal set of equivalent elements

Equivalent Classes: Example

- Suppose R = {(1, 11), (7, 11), (2, 12), (12, 8), (11,12), (3, 13), (4,13), (13, 14), (14, 9), (5, 14), (6, 10)} and n = 14
  - For simplicity omit reflexive and transitive pairs

- Three equivalent classes
  - {1, 2, 7, 8, 11, 12}
  - {3, 4, 5, 9, 13, 14}
  - {6, 10}

Equivalence Class Problem

- Determine the equivalence classes

- The offline equivalence class problem
  - Given n elements and Given a relation R
  - We are to determine the equivalence classes
  - Can be solved easily with various ways

- The online equivalence class problem (namely, the Union-Find problem)
  - R is built incrementally by online inputs
  - Begin with n elements, each in a separate equiv class
  - Process a sequence of the operations
    - combine(a, b) : combine an equiv class A and an equiv Class B
    - find(theElement) : find a class having theElement

Combine and Find Operation

- combine(a,b)
  - Combine the equivalence classes that contain elements a and b into a single class
  - Is equivalent to

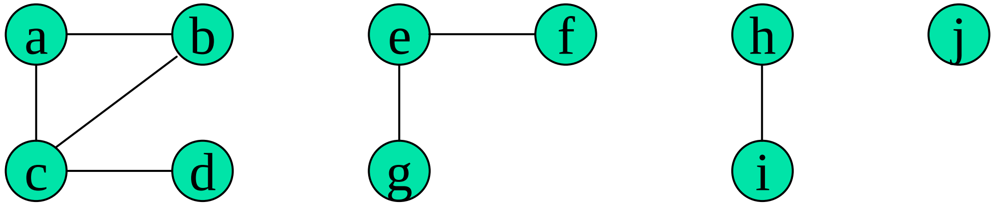
    classA = find(a);

    classB = find(b);

    if (classA != classB)   union(classA, classB);

- find(theElement)
  - Determine the class that currently contains element theElement
  - To determine whether two elements are in the same class

Union-Find Problem Example



Edge processed	Collection of disjoint sets
initial sets	{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}
(b, d)	{a} {b, d} {c} {e} {f} {g} {h} {i} {j}
(e, g)	{a} {b, d} {c} {e, g} {f} {h} {i} {j}
(a, c)	{a, c} {b, d} {e, g} {f} {h} {i} {j}
(h, i)	{a, c} {b, d} {e, g} {f} {h, i} {j}
(a, b)	{a, b, c, d} {e, g} {f} {h, i} {j}
(e, f)	{a, b, c, d} {e, f, g} {h, i} {j}
(b, c)	{a, b, c, d} {e, f, g} {h, i} {j}

We are given set of elements and build up equivalence classes

At each step, sets are build by find and union operations

Equiv Class Applications – Machine-scheduling problem (1)

- How to make a feasible schedule?
  - A single machine that is to perform n tasks
  - Each task has release time and deadline and is assigned to a time slot between its release time and deadline
- Example

Task	A	B	C	D
ReleaseTime	0	0	1	2
Deadline	4	4	2	3

```
  0     1     2     3      4
-----------------------------------→
      A    C     D      B
```

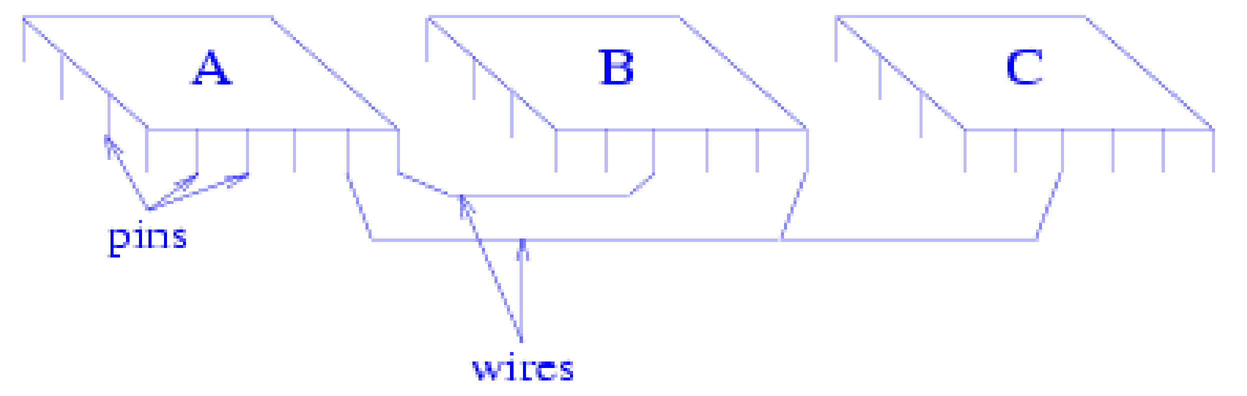
Equiv Class Applications – Machine-scheduling problem (2)

- Method to construct a schedule
  1. Sort the tasks into nonincreasing order of release time
  2. For each task, determine the free slot nearest to, but not after, its deadline
     - If this free slot is before the task’s release time, fail
     - Otherwise, assign the task to this slot

Equiv Class Applications – Machine-scheduling problem (3)

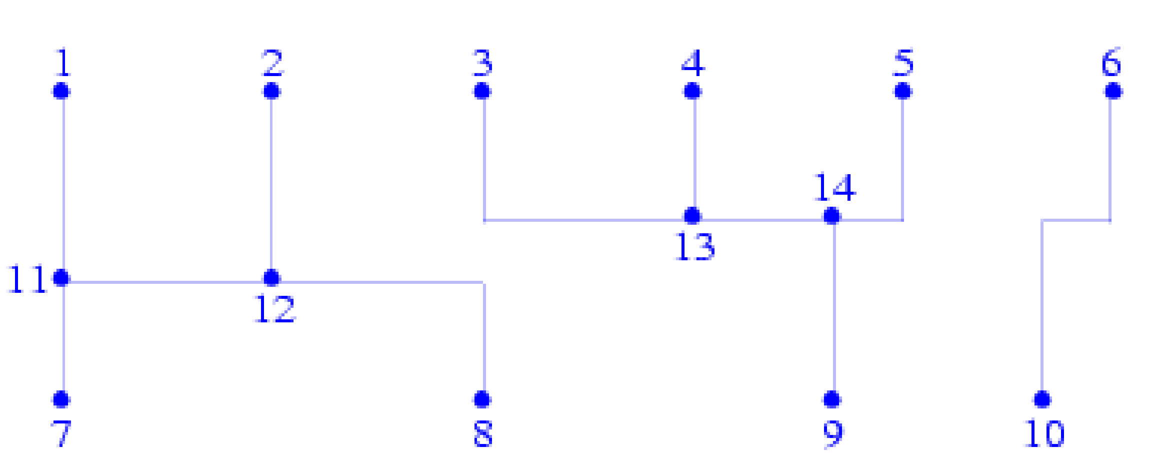
- The online equivalence class problem can be used to implement step(2)
  - near(a) : the largest i such that i<=a and slot i is free
  - If no such i exists, near(a) = near(0) = 0
  - Two slots a and b are in the same equivalence class iff near(a) = near(b)
  - Initial condition : near(a) = a for all slots, and each slot is in a separate equivalence class
  - When slot a is assigned a task in stop(2), near changes for all slots b with near(b) = a
    - When slot a is assigned a task, perform a union on the equivalence classes that currently contain slots a and a - 1

Equiv Class Applications – Circuit-wiring Problem (1)



- Electrically equivalent
  - Connected by a wire or there is a sequence of pins connected by wires
- Net
  - Maximal set of electrically equivalent pins

Equiv Class Applications – Circuit-wiring Problem (2)



- Each wire may be described by the two pins that it connects
- Set of wires {(1,11), (7,11),…, (6,10)}
- Nets {1,2,7,8,11,12},{3,4,5,9,13,14},{6,10}

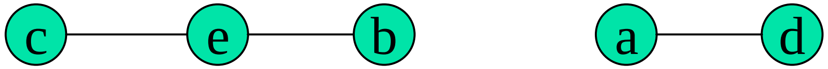
Equiv Class Applications - Circuit-wiring Problem (3)

- The Offline net finding problem
  - Given the pins and wires
  - Determine the nets
  - Modeled by the offline equivalent problem with each pin (as a member of U) and each wire (as a member of R)
- The Online net finding problem
  - Begin with a collection of pins and no wires
  - Perform a sequence of operations of the form
    - Add a wire “one-by-one” to connect pins a and b
    - Find the net that contains pin a

OECP: The 1st Union-Find solution

- By array equivClass[]
- equivClass[i] is the class that currently contains element I
- Inputs to Union are equivClass values
- Initialize & Union → O(n), Find → O(1)
- Given n elements: 1 initialization, u unions, and f finds → O(n + u*n + f)

OECP using Arrays



		a	b	c	d	e	
Initial State		1	2	3	4	5	Classes with one element each
combine(b,c)		1	2	2	4	5	‘a’ and ‘b’ belongs to same class
combine(b,e)		1	2	2	4	2	
combine(a,d)		1	2	2	1	2	{a, b, c} and {d, f}

※index 0 is not used

OECP: The 1st Union-Find Solution (1)

```
public class UnionFindFirstSolution  {
      static int [] equivClass;
      static int n;     // number of elements

      // initialize numberOfElements classes with one element each
      static void initialize(int numberOfElements)  {
            n = numberOfElements;
            equivClass = new int [n + 1];
            for (int e = 1; e<=n; e++)
                  equivClass[e] = e;
      }
    // continued
```

OECP: The 1st Union-Find Solution (2)

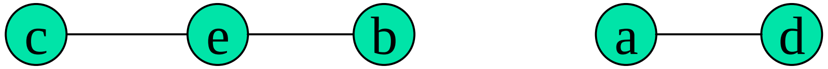
```
// unite the classes classA and classB
 static void union(int classA, int classB) {
       // assume classA != classB
       for (int k = 1; k <= n; k++)
             if (equivClass[k] == classB)
                   equivClass[k] = classA;
 }

 // find the class that contains theElement
 static int find(int theElement) {
       return equivClass[theElement];
 }
}
```

The 2nd Union-Find Solution

- Reduce the time complexity of the union operation by keeping a chain for each equivalence class
    - We can find all elements in a given equivalence class by going down the chain
    - Size and Next are added
    - In array, full scan is required for changing a class

The 2nd Union-Find Solution using Chains



			a	b	c	d	e
Initial State	equivClass	0	1	2	3	4	5
	size	0	1	1	1	1	1
	next	0	0	0	0	0	0
combine(b,c)		0	1	3	3	4	5
		0	1	1	2	1	1
		0	0	0	2	0	0
combine(c,e)		0	1	3	3	4	3
		0	1	1	3	1	1
		0	0	5	2	0	0
combine(a,d)		0	4	3	3	4	3
		0	1	1	3	2	1
		0	0	0	0	1	0

The 2nd UFS: The Class EquivNode

```
class EquivNode
{
     int equivClass;  // element class identifier
     int size;        // size of class
     int next;        // pointer to next element in class

    // constuctor
     EquivNode (int theClas, int theSize) {
           equivClass = theClass;
           size = theSize;
           // next has the default value 0
     }
}
```

The Class UnionFindSecondSolution (1)

```
public class UnionFindSecondSolution  {
    static EquivNode [] node;          // array of nodes
    static int n;                      // number of elements

    // initialize numberOfElements classes with one element each
    static void initialize(int numberOfElements)  {
        n = numberOfElements;
        equivClass = new EquivNode[n + 1];
        for (int e = 1; e<=n; e++)
            // node[e] is initialized so that its equivClass is e
            node[e] = new EquivNode(e, 1);
    }
  // continued
```

The Class UnionFindSecondSolution (2)

```
static void union(int classA, int classB) {
// assume classA != classB, make classA smaller class
if (node[classA].size > node[classB].size) { // swap classA and classB
          int t = classA;       classA = classB;       classB = t; }
int k;
for (k = classA; node[k].next != 0; k = node[k].next)
          node[k].equivClass = classB;
 node[k].equivClass = classB;
 // insert chain classA after first node in chain classB and update new chain size
 node[classB].size += node[classA].size;
 node[k].next = node[classB].next;
 node[classB].next = classA;
}

static int find(int theElement) {
               return node[theElement].equivClass;
     }
}
```

Summary (1)

- Simulated-pointer representation
  - What if we want to have linked structures on disk
  - What if we want to have user-defined pointers instead of Java references
  - Simulated pointers are represented by integers rather than by Java references
- To use simulated pointers
  - Must implement our own memory management scheme: a scheme to keep track of the free nodes in our memory (i.e., array of nodes)

Summary (2)

- Simulated Pointers
- Memory Management
- Comparison with Garbage Collection
- Simulated Chains
- Memory Managed Chains
- Application: Union-Find Problem